NPS-CS-10-002

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Specification and Validation of Space System Behaviors

By Doron Drusinsky and Steven Raque

02 February 2010

Approved for public release; distribution is unlimited

Prepared for: NASA IV&V Facility
100 University Drive
Fairmont, WV 26554

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

This report was prepared for the NASA IV&V Facility and funded by the NASA IV&V Facility.

This report was prepared by:

_____
Doron Drusinsky
Associate Professor of Computer Science
Naval Postgraduate School

Reviewed by:

_____
Peter J. Denning, Chairman
Department of Computer Science

Released by:

_____
Karl A. van Bibber
Vice President and
Dean of Research

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
02-02-2010	Technical Report	Jan. 2009 – Dec. 2009

4. TITLE AND SUBTITLE
Specification and Validation of Space System Behaviors

5a. CONTRACT NUMBER
NNG07LD021

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Doron Drusinsky
Steven Raque

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

NPS-CS-10-002

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NASA IV&V Facility
100 University Dr.
Fairmont. WV 26554

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The NASA Independent Verification and Validation (IV&V) Facility is using formal specification techniques for the IV&V of the flight software for several upcoming missions. Such formal specifications are typically created on the basis of natural-language (NL) requirement specifications that are formalized at a later stage. This paper describes a technique for the discovery of NL requirements by systematic analysis of UML Activity Diagrams and Sequence Diagrams that represent critical mission operational scenarios and sequences. Our technique demonstrates a pattern oriented approach where patterns of NL requirements are mined out of the UML models based on predetermined categories of assertions, including Bounded Eventualities, Loops, Reentrance, and Order and Precedence.

15. SUBJECT TERMS
Validation and verification, formal methods, specification, UML, scenarios, requirements, patterns

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Doron Drusinsky
U	U	U	SAR	32	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

1. Introduction

Formal requirement specifications are mathematically rigorous specifications that are readable and executable by a computer based verification system. Many formal specification languages have been described over the past two decades, including linear-time temporal logic (LTL) [LTL], branching-time temporal logics [CTL], and more recently, statechart assertions [Dr1]. For most NASA applications, the ultimate purpose of formal specifications is to be applied towards subsequent computer-aided verification, using techniques such a model-checking, theorem proving, and run-time verification; [DMS] provides a three dimensional comparison of these three primary verification techniques.

Harel statecharts [Ha], currently part of the UML standard, are typically used for design analysis and implementation. In his 2006 book [Dr1], the author describes the application of deterministic and non-deterministic statecharts-assertions to formal requirement specification and run-time verification. This approach is currently in active use by the NASA IV&V facility.

Runtime Verification (RV) is a class of methods for tracking the temporal behavior of an underlying application and comparing it against its formal specification. RV methods range from off-line logging methods to on-line tracking of complete formal requirements. NASA is currently using an log file based monitoring approach to verify statechart assertion specifications for GPM and the Crew Vehicle component of the Orion mission.

Effective use of RV-based verification depends on the construction of sound and complete correctness property assertions. *Validation* is the process of assuring the correctness of formal specification assertions with respect to their informal intent as manifested by natural language specifications. In [DKS], Drusinsky, Kadir, and Shing describe the validation process used to create sound LTL/MTL correctness properties. In [DOS] Drusinsky, Otani, and Shing describe a set of validation scenario templates, or patterns, that are common to most statechart assertion validation test suites.

This paper describes a process for extracting natural language specifications from informal UML diagrams [UML]. Hence, it addresses one of the most difficult concerns for many formal specification and verification techniques that depend on the existence of good natural language specifications. In fact, this also addresses a concern of old fashioned human based testers as well; after-all, human testers need meaningful, accurate, and detailed natural specifications in order to write good tests.

To this end, this paper describes a systematic process for obtaining NL requirements from a UML based System Reference Model (SRM). The paper describes a case study conducted at the NASA IV&V facility in which we created: (i) created SRM's for the Gravity Recovery and Interior Laboratory (GRAIL) mission and the Global Precipitation Measurement (GPM) mission, (ii) extracted natural language requirements from each SRM, (iii) created statechart assertion formal specifications for those requirements, and (iv) performed validation to assure that our formal specifications correctly represent the spirit behind the natural language specifications.

This paper is organized as follows. Section 2 describes the GRAIL and GPM missions while sections 3 through 5 describe our results using GRAIL as an example: section 3 describes the SRM creation process, section 4 describes the process of extracting NL requirements from the SRM, and section 5 describes four categories of requirements that can often be extracted from an SRM.

We used the StateRover Eclipse plug-in to create statechart assertions [Ec, Sr] and JUnit [Ju] (the de-facto standard framework for Java testing) as a framework for validation.

2. GRAIL and GPM Missions

GRAIL's mission is to reveal the moon's internal structure and evolution by flying twin spacecraft orbiters in tandem orbits around the moon for several months to measure its gravity field in unprecedented detail [GRAIL]. The orbiters will constantly adjust attitude to face each other, while the distance between the orbiters is expected to fluctuate in response to changes in lunar gravitation field.

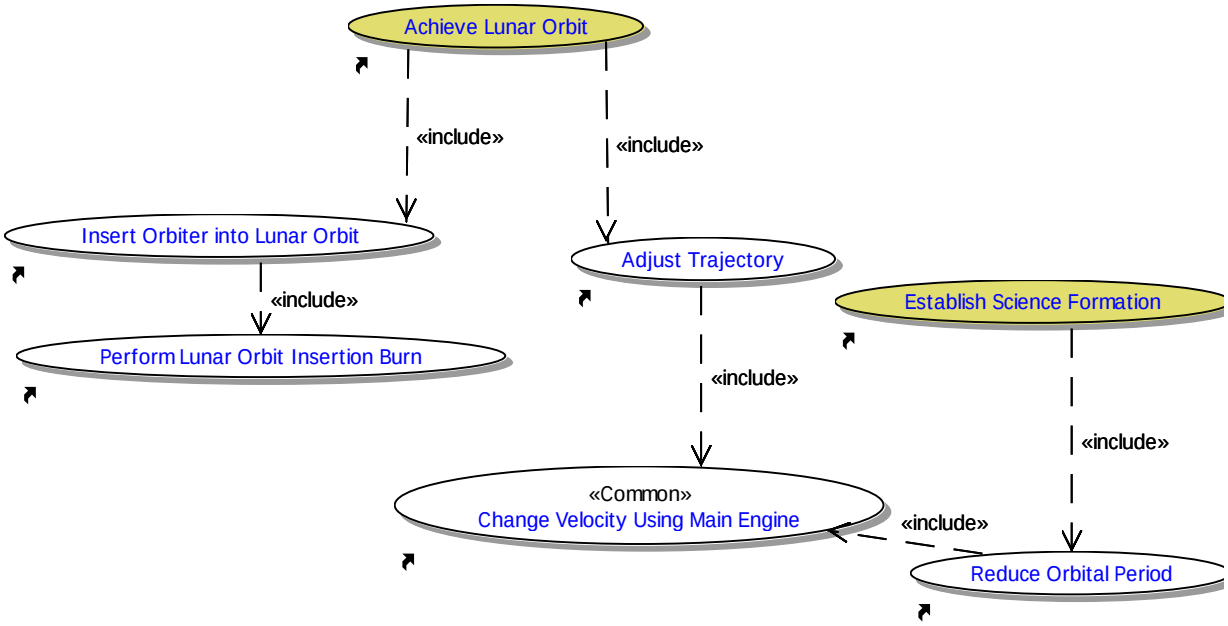
The GPM mission will study global precipitation (rain, snow, and ice) from low earth orbit [GPM]. Carrying a dual frequency radar instrument and a passive microwave radiometer, the GPM spacecraft will serve as a calibration standard, and each instrument requires precise pointing for target location determination.

GRAIL and GPM, as well as many other scientific missions can be thought of as two separate missions: the flight mission and the scientific mission. The flight-mission is the mission we use in this paper. GRAIL's main high-level use-case scenario is to send the pair of orbiters into pre-prescribed lunar orbits; the orbiters will constantly adjust attitude so to face each other, while the distance between the orbiters is expected to fluctuate in response to changes in lunar gravitation field. The orbiters will measure the fluctuating distance as part of the scientific mission

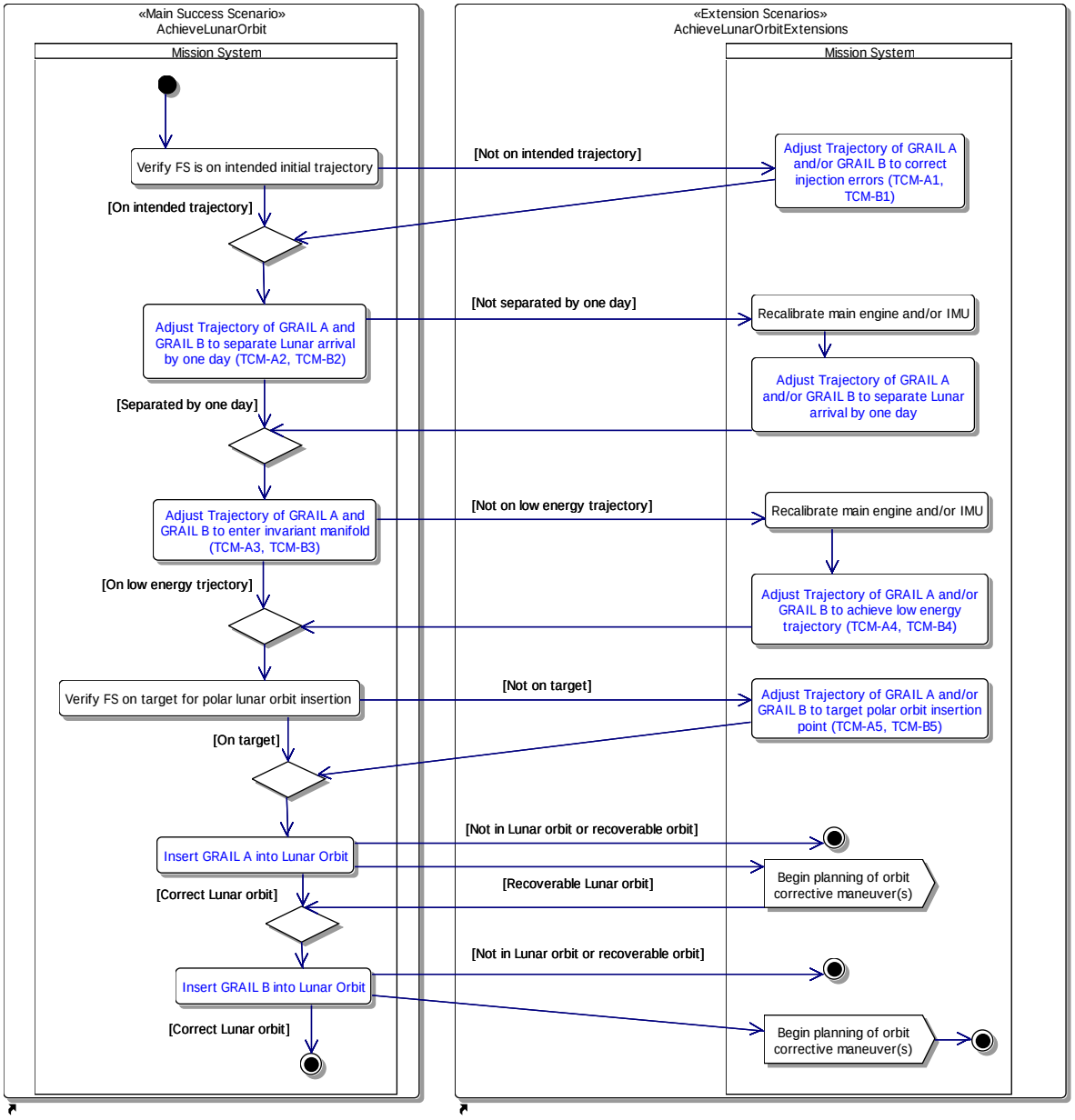
3. GRAIL SRM: The Process

As stated above, the SRM is a way of capturing the system functional architecture required to achieve the mission goals. At the highest level of abstraction, a use case diagram is used to show all the goals and the functions necessary to achieve the goals. Fig. 1a is part of a top-level use case diagram representing functional goals for the GRAIL mission. As is common with use case driven modeling, Fig. 1a does not reflect the only effective way to parse the system goals and functionality. Rather, the figure is intended to give a sense of the overall approach.

From this level, the SRM is elaborated with textual use cases that include preconditions, postconditions, a trigger, a main success scenario, and extension scenarios. Each of these use cases are represented as AD's (as depicted in Fig. 1b), and critical actions (use case steps) are further elaborated with more AD's and SD's. In this approach, the UML diagrams take the place of the Functional Flow Block Diagrams (FFBDs). One key advantage over FFBDs is that UML (and alternatively SysML) provide more diverse constructs that can represent diverse perspectives on required system behavior and functionality. Because of their similarity to flowcharts, it has been found that AD's are a broadly understandable way of capturing mission scenarios.



a. Top level use case diagram depicting two of eight high-level goals/functions for the GRAIL mission.



b. *AchieveLunarOrbit* activity diagram representing the mission scenario that starts with both orbiters being separated from the launch vehicle and being on an initial trajectory departing earth orbit, and concluding with both orbiters in lunar orbit.

Figure 1. High level SRM view of the GRAIL flight mission

4. GRAIL: From SRM to NL Requirements

The GRAIL SRM consists of use-case diagrams, use-cases, activity diagrams (AD's), sequence diagrams (SD's), and finite state machines (FSM). These artifacts are informal and as discussed below are not usable out of the box for subsequent verification. Stated differently, the SRM artifacts are almost recommendations, and are therefore not definitive "shall or "shall not" requirements; this point will be demonstrated using some of the examples the follow. Hence enters the next important step in our process namely, *formal specification requirement and assertion extraction*. Our strategy is to first identify *items of concern*, or *concerns* for short, by observing the GRAIL SRM diagrams in a certain critical manner. It is important to note here that a concern is not about the quality of a GRAIL AD, SD, or FSM as such; rather, it is a concern or constraint about the GRAIL flight code behavior as prescribed by the AD, SD, or FSM that a reasonable tester should address in his/her test harness. Stated differently, it is because we have well written and informative, albeit informal, SRM diagrams that it is possible to identify test related concerns from them. In a later step, we will create computer-based assertions, they too being testers of sorts, to address those concerns during verification.

4.1 The Assertion Development Process

We will create concern related assertion in a two step process. The first step is to identify a concern in an AD, SD, or FSM and capture it as a NL requirement. The second step is to formalize a corresponding statechart-assertion (for a reactive requirement) or propositional-assertion (for a transformational requirement) and to validate it using the process described earlier.

To simplify the first step, we have identified four categories of concerns, numbered C1 through C4 For each category we will present one or more examples that consist of the UML diagram, a narrative description of concerns, the resulting NL requirements, and their corresponding assertions and validation tests.

5. Categories of Concerns

5.1. C1: Bounded Eventualities

The AD of Fig. 2a captures GRAILs flight behavior as of the separation of the two orbiters from the primary launch vehicle, and until they are inserted into their respective lunar orbits. The AD of Fig. 2b details the *Perform Lunar Orbit Insertion Burn* (*PerformLOIBurn*) activity of Fig. 2a, an activity performed by each orbiter, intended to insert the orbiter in a lunar orbit prescribed by MS. The SD of Fig. 2c depicts the same scenario[1].

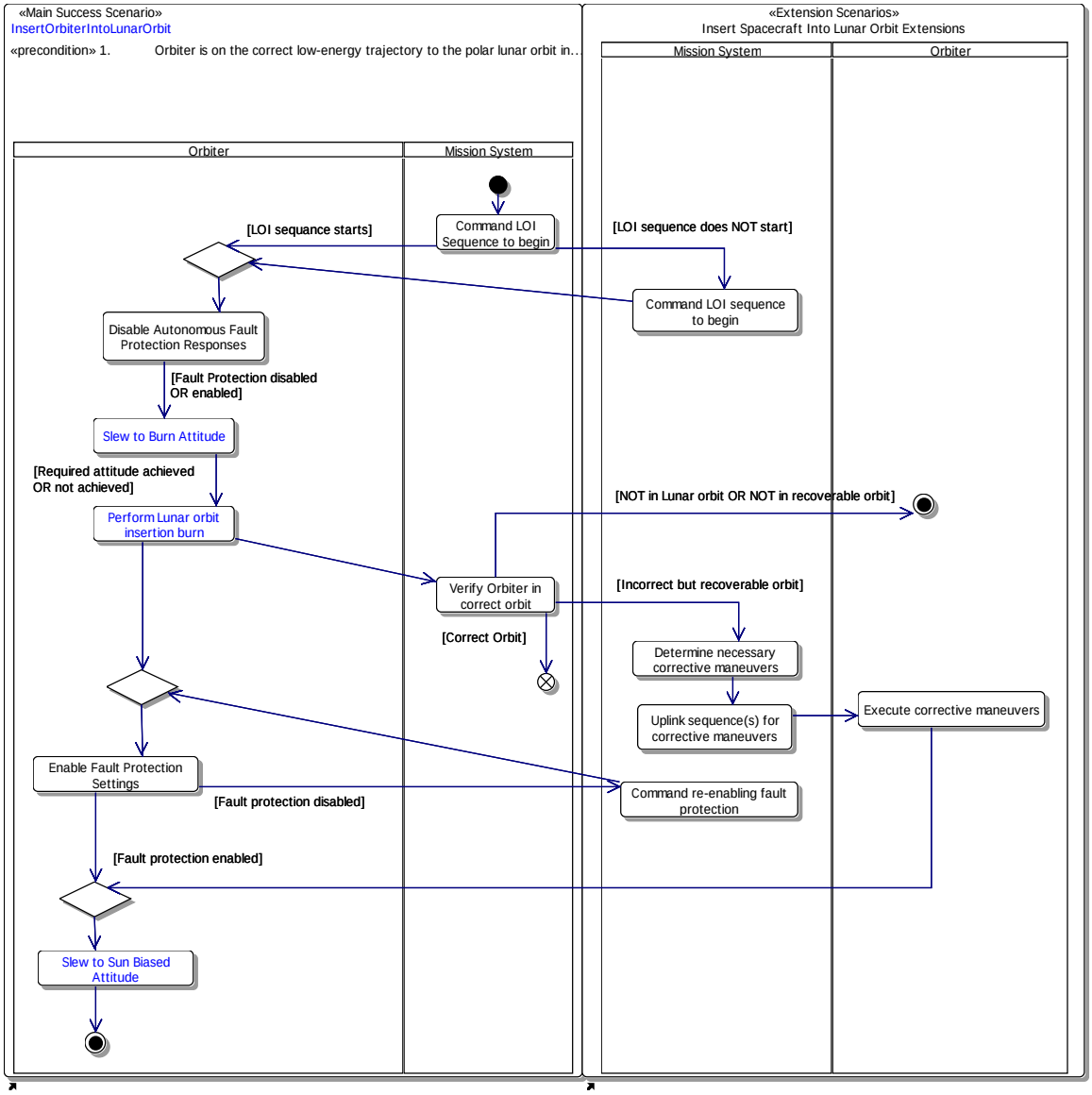
The insertion of an orbiter into the correct lunar orbit is performed by turning on the Propulsion Fuel Valve at the right time for a specified duration in a reverse boost mode, i.e., in a direction opposite to the acceleration vector. Mission Systems (MS) control determines and uploads those parameters in a command sequence. The orbiter is meant to

[1] One could argue that the SD view is preferable for our purpose because SD messages are easier to affiliate with method names used in assertions and in the executable-SRM section below.

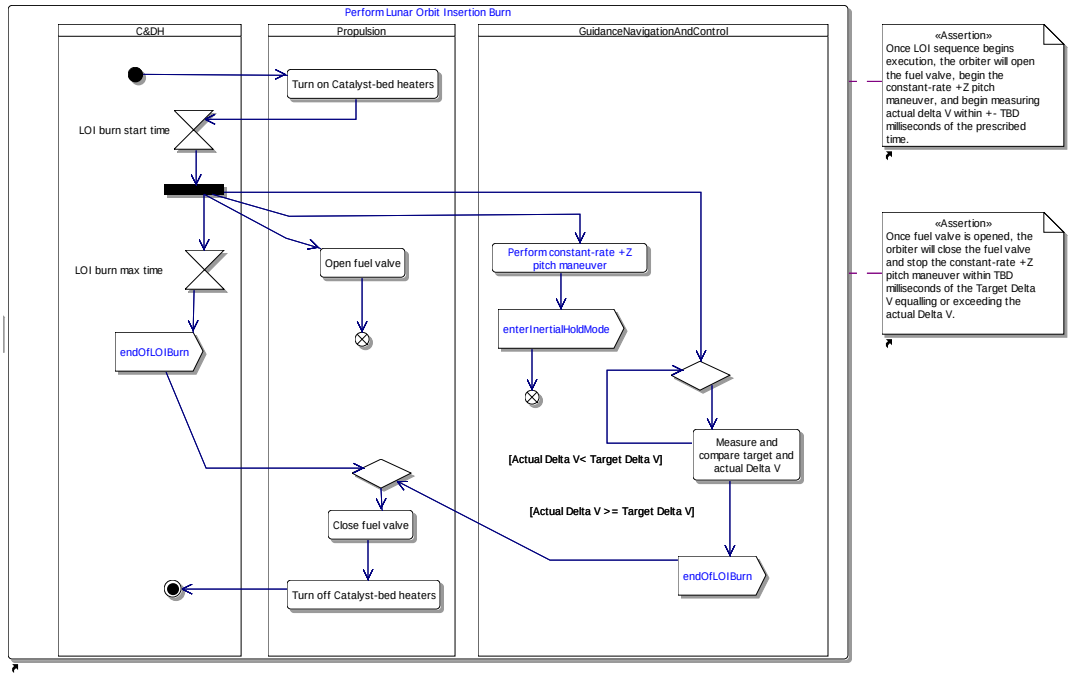
be autonomous during this boost phase; this is because the boost operation is time sensitive, i.e., no external entity should be allowed to interfere in the operation, not even MS or fault protection. This is indicated by the AD of Fig. 2a, where fault protection is turned off before the *slew* and *burn* activities and turned back on afterwards. A concern here is therefore that the orbiter performs boost according to parameters prescribed by MS the command sequence, up to some level of acceptable tolerance. Hence, the NL-requirement is R1: *once LOI burn sequence is uploaded, the orbiter will, within the time prescribed in the command sequence parameters (plus/minus* Δ_{t1}*), perform a burn for the duration prescribed in the command sequence parameters (plus/minus* Δ_{t2}*).*

Clearly, the key concern of R1 is with tasks performed within a prescribed time, i.e., with *bounded eventualities*.

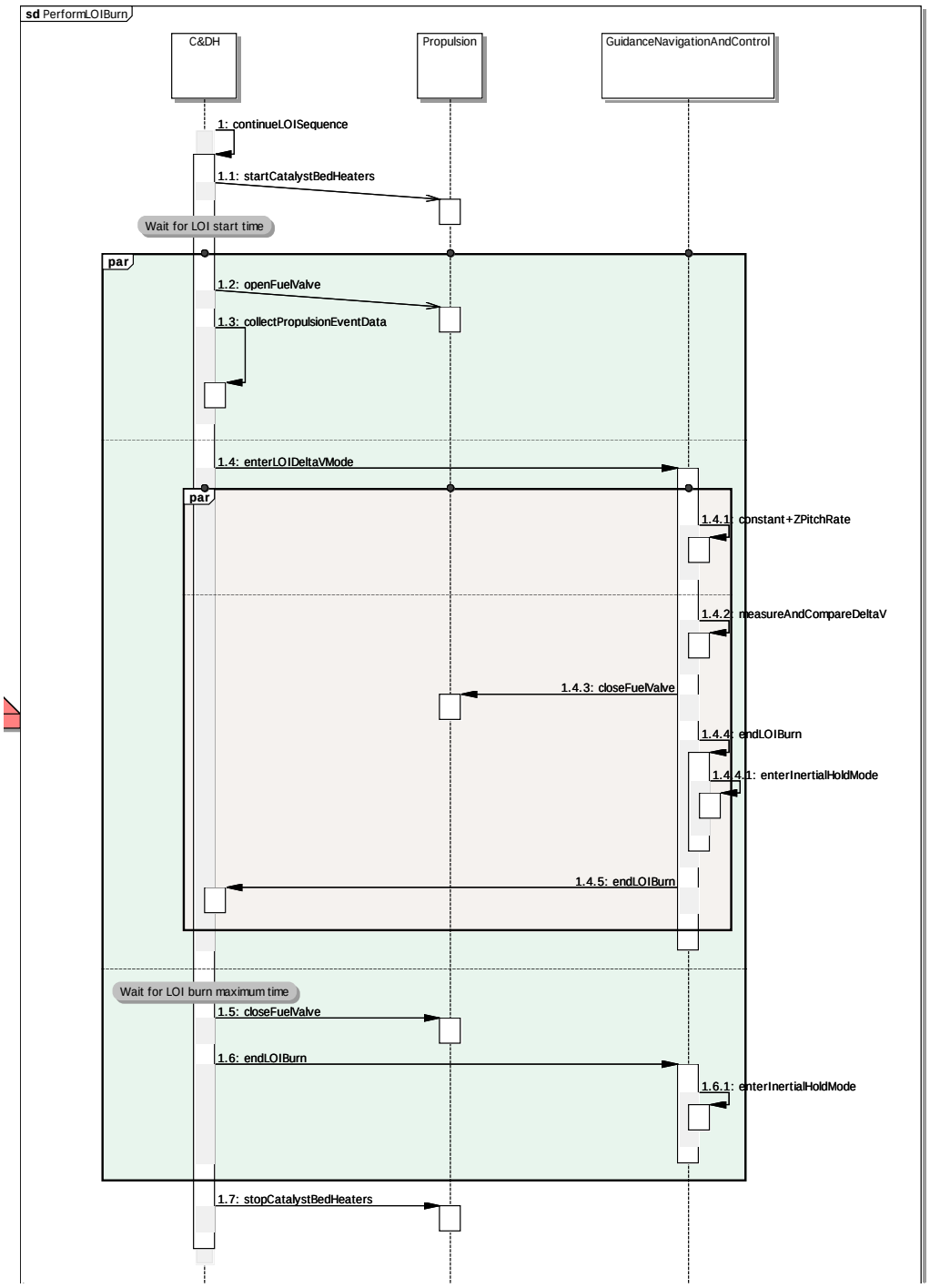
Fig. 3 depicts the statechart-assertion for requirement R1.



a. *InsertOrbiterIntoLunarOrbit* activity diagram



b. *PerformLOIBurn* activity diagram.



c. *PerformLOIBurn* sequence diagram.

Figure 2.

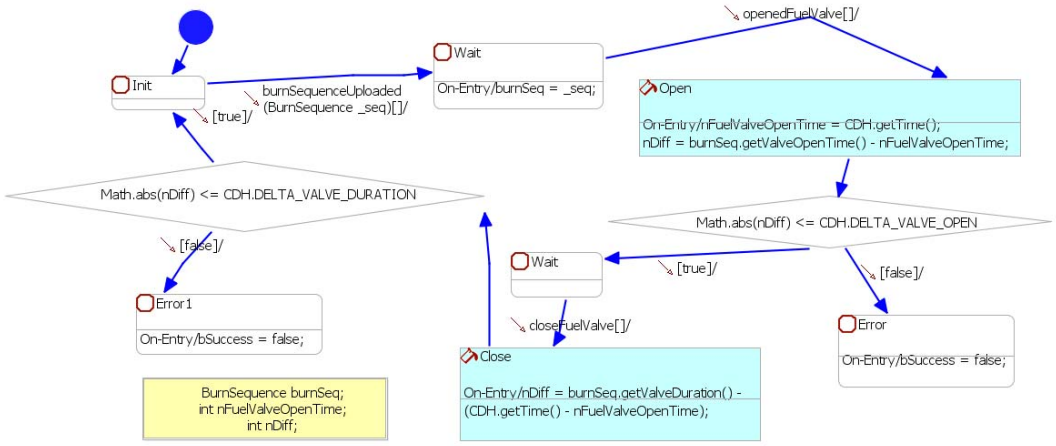


Figure 3. statechart-assertion for NL requirement R1.

Listing 1 includes two validation tests for this assertion. Note that the tests create their own mockup accessory *BurnSequence* data objects.

```
public void testMe() {
      int nTime= 0;
      BurnSequence seq = new BurnSequence(1000, 100);
      assertion.burnSequenceUploaded(seq);
      nTime = 1009;
      CDH.setTime(nTime);
      assertion.openedFuelValve();
      nTime += 96;
      CDH.setTime(nTime);
      assertion.closeFuelValve();
      nTime += 100;
      CDH.setTime(nTime);
      assertTrue(assertion.isSuccess());
}
```

a. A success validation test scenario.

```
public void testMe() {
      int nTime = 0;
      BurnSequence seq = new BurnSequence(1000, 100);
      assertion.burnSequenceUploaded(seq);
      nTime += 1008;
      CDH.setTime(nTime);
      assertion.openedFuelValve();
      nTime += 94;
      CDH.setTime(nTime); // too early
      assertion.closeFuelValve();
      nTime += 100;
      CDH.setTime(nTime);
      assertFalse(assertion.isSuccess());
}
```

b. A failure validation test scenario.

Listing 1. Validation tests for the statechart-assertion of Fig. 3.

Note that the *Bounded Eventualities* category is actually a subset of the *Order and Precedence* category; hence, Fig. 2a and Fig. 2b will be analyzed again in that section.

5.2. C2: Loops

This category analyzes possible system behavior issues related to loops found in AD's, SD's, or FSM's.

The *AttitudeControl* FSM of Fig. 4 depicts the three primary modes of attitude control:

1. *Slew*, where reaction wheels are used to adjust attitude, if associated forces are sufficiently small.
2. *LOIDeltaV* - Lunar Orbit Insertion Delta-V, where .
3. *InertialHold,* The no-operation or idle mode.

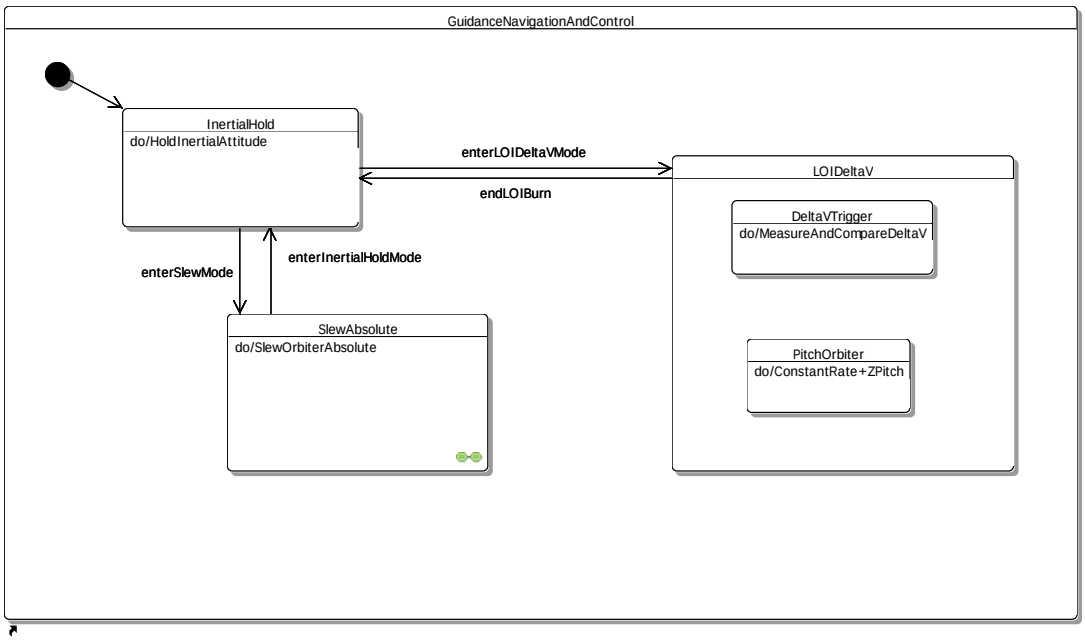


Figure 4. FSM for Attitude Control modes.

Clearly, this FSM implies the following requirement; R2: *The system cannot change modes from Slew to LOIDeltaV or visa-versa without being in the InertialHold mode for at least* T *seconds*. A statechart assertion for this requirement is depicted in Fig. 5. Listing 2 contains a validation test for this assertion.

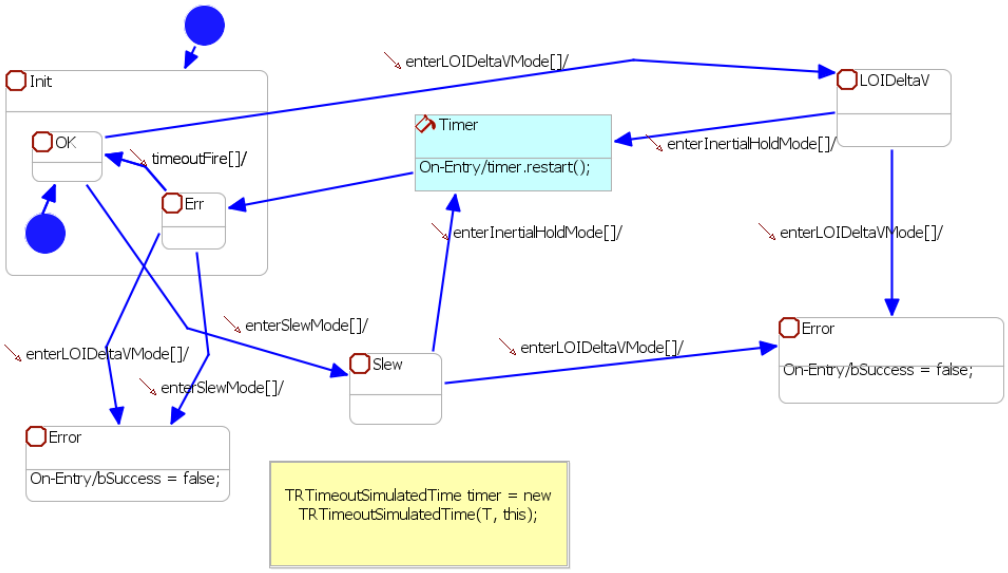


Figure 5. Statechart-assertion for requirement R2.

```
public void testMe() {
      assertion.enterLOIDeltaVMode();
      assertion.incrTime(20);
      assertion.enterInertialHoldMode();
      assertTrue(assertion.isSuccess());
      assertion.incrTime(35);
      assertion.enterSlewMode();
      assertion.incrTime(10);
      assertTrue(assertion.isSuccess());
      assertion.enterInertialHoldMode();
      assertion.incrTime(10);
      assertion.enterSlewMode();
      assertFalse(assertion.isSuccess());
}
```

Listing 2. A validation test for the statechart-assertion of Fig. 5 .

Note that although R2 is not a looping requirement per-se, it helped uncover the following looping concern: *should there be a limit on the number of times or the frequency in which mode can change between Slew and LOIDeltaV, or visa-versa*? The authors sulution yielded the following requirement. R3 : *The system can toggle between Slew and LOIDeltaV modes at most twice per minute.* A (non-deterministic) statechart assertion for this requirement is depicted in Fig.6. Listing 3 contains a validation test for this assertion.

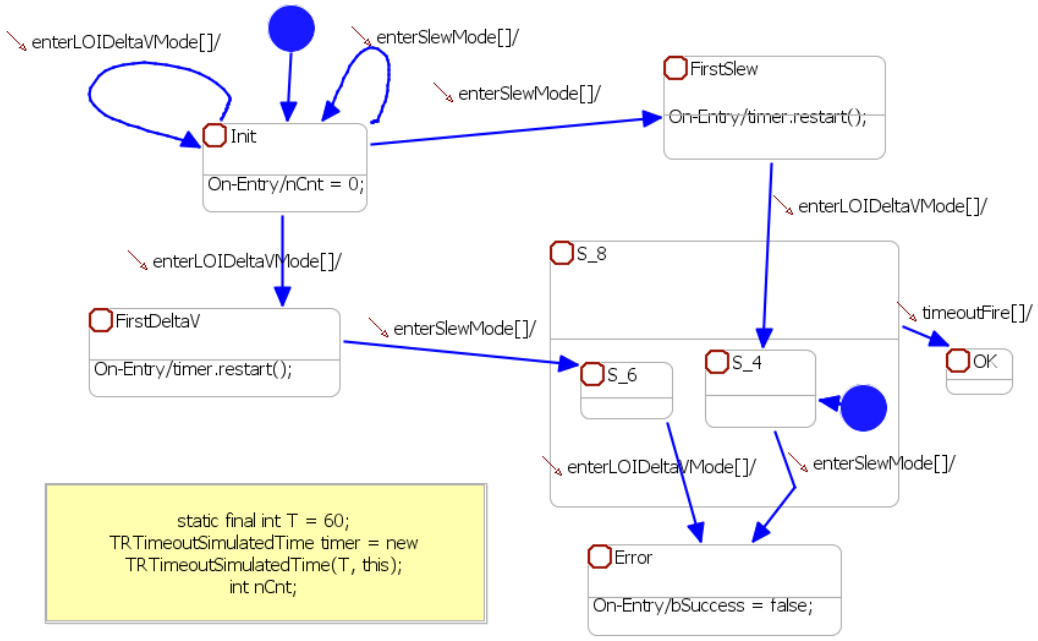


Figure 6. A (non-deterministic) statechart-assertion for requirement R3 .

```
public void testMe() {
      assertion.enterSlewMode();
      assertion.incrTime(20);
      assertion.enterLOIDeltaVMode();
      assertion.incrTime(50);
      assertion.enterSlewMode();
      assertTrue(assertion.isSuccess());
      assertion.incrTime(4);
      assertion.enterLOIDeltaVMode();
      assertFalse(assertion.isSuccess());
}
```

Listing 3. A validation test for the statechart-assertion of Fig. 6 .

Consider the *Slew Orbiter Absolute* AD of Fig. 8, a refinement of the *Slew to Burn* activity within the *Adjust* Trajectory AD of Fig. 7. *The Slew Orbiter Absolute* activity performs a slew to an absolute attitude. After selecting reaction wheels as the actuator that physically induces the slew, it simultaneously predicts and determines attitude. If either of those activities has an error then the extension side of the AD shows that the Kalman filter[2] is reset and the loop is then closed with the two activities being executed again. A looping concern here is: *how many times and how often this loop can be*

[2] The Kalman filter is an efficient recursive filter that estimates the state of a linear dynamic system from a series of noisy measurements. See http://en.wikipedia.org/wiki/Kalman_filter

traversed? With the SME's feedback we forged the following requirement R4: *whenever the Kalman Filter is reset more than* N *times in a* 5 *minute interval then Safe mode should be entered within* 30 *seconds afterwards* .

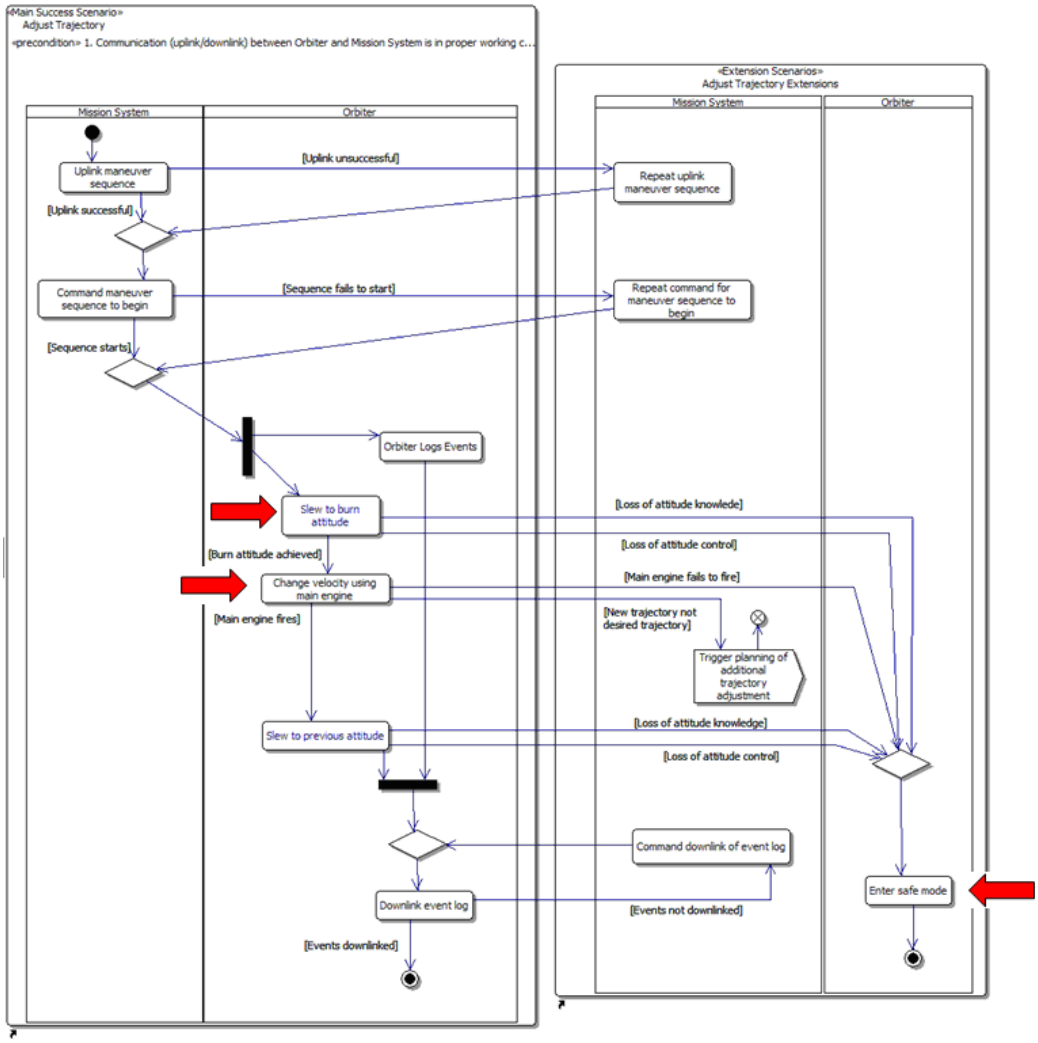


Figure 7. The *AdjustTrajectory* activity diagram

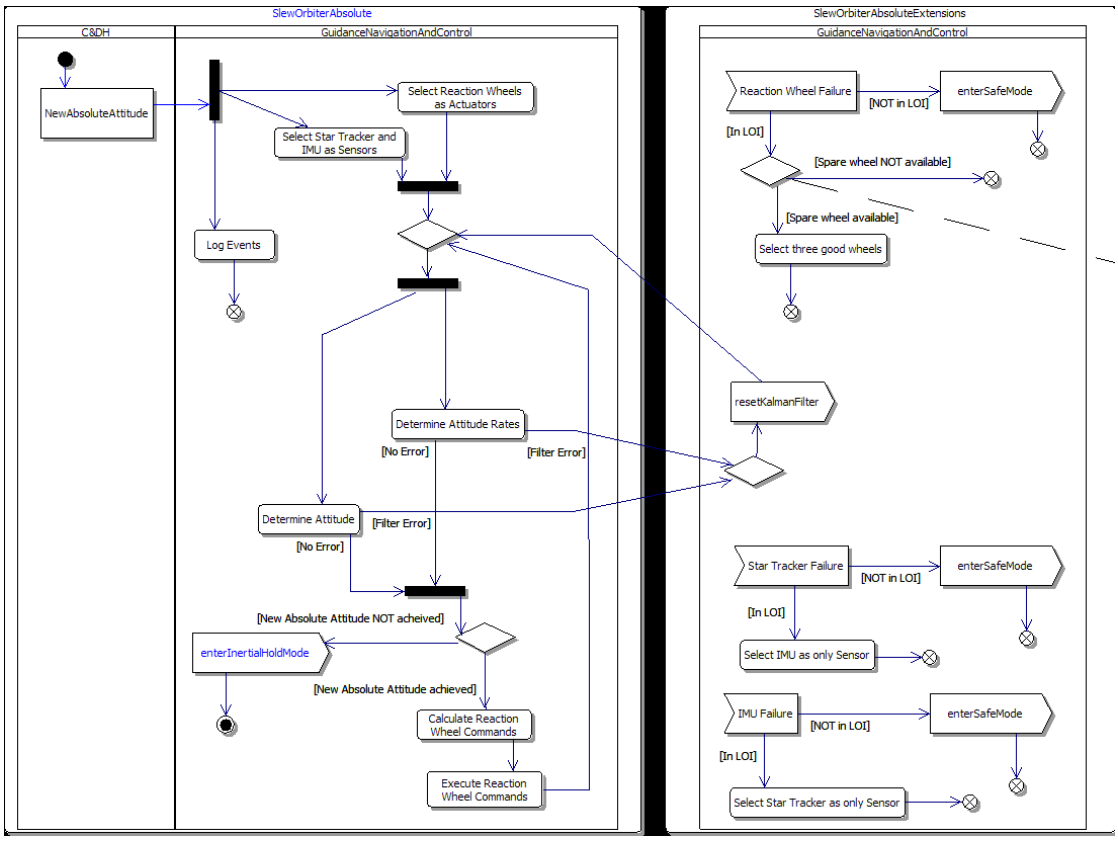


Figure 8. *Slew Orbiter Absolute* activity diagram, a refinement of the *Slew to Burn* activity of Fig. 4.7.

A (non-deterministic) statechart-assertion for this requirement is depicted in Fig. 9. Listing 4 is a corresponding validation test.

Another looping concern associated with the AD of Fig. 8 is related to entering and ending *Safe Mode*, as follows. It is fault protection that forces the orbiter into safe mode, and it is a human in the loop in MS that has the authority and ability to bring the orbiter out of that mode. The concern is therefore whether there should be a limit on the number and frequency of such switches in and out of safe mode. Indeed, a similar a problem had occurred in the past. Since requirement R3 (and its associated statechart-assertion of Fig. 6) is similar in nature to this requirement we will not discuss it further.

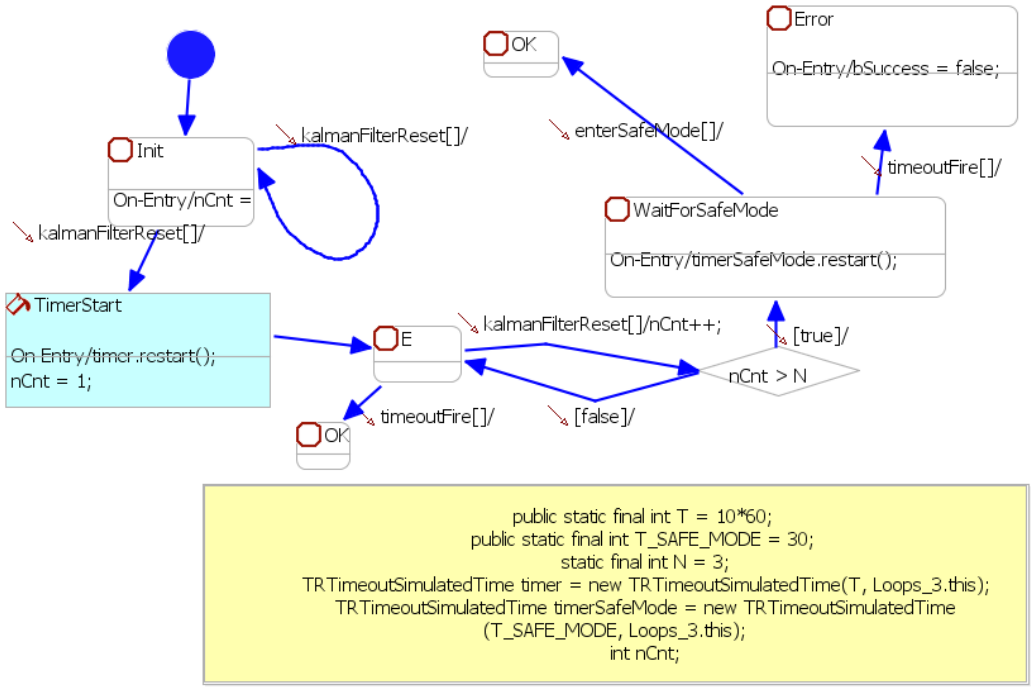


Figure 9. A (non-deterministic) statechart-assertion for requirement R4.

```
public void testMe() {
      // Adapting the validation test of Listing 3.7:
      // T/30 is the ratio between the time constraint in
      // Listing 3.7 and here
      int nFACTOR = 10*60/30;

      assertion.kalmanFilterReset();
      assertion.incrTime(6*nFACTOR);
      assertion.kalmanFilterReset();
      assertion.incrTime(5*nFACTOR);
      assertion.kalmanFilterReset();  // <=- start of violating
                                      // sequence
      assertion.incrTime(20*nFACTOR);
      assertion.kalmanFilterReset(); // 4'th as of time 0 happens
                                     // at time 31*nFACTOR --> ok
      assertion.incrTime(6*nFACTOR);
      assertion.kalmanFilterReset(); // 4'th as of time 6 after
                                    //addition 31*nFACTOR --> ok
      assertTrue(assertion.isSuccess());
      assertion.incrTime(1*nFACTOR);
      assertion.kalmanFilterReset(); // 4'th E as of time
                                      //11*nFACTOR happens after
                                     // additional 27 time units
      assertion.incrTime(31);
      assertion.enterSafeMode(); // too late
      assertFalse(assertion.isSuccess());
}
```

Listing 4. A validation test for the statechart-assertion of Fig. 9.

5.3. C3: Reentrance

This category encapsulates various concerns about the system performing a scenario that effectively reenters an activity as specified by some UML AD, SD, or FSM diagram, before it has completed its current invocation.

When the authors first analyzed reentrance issues for the *PerformLOIBurn* activity of Fig. 2b, we immediately realized that re-entrance was prohibited. In fact, although a orbiter might have a plurality of command sequences active at the same time, at most one of those can be a Propulsion Burn sequence. Hence enters requirement R5 : *at most one propulsion burn sequence (per orbiter) can be active at any given time*. The statechart-assertion for this requirement is depicted in Fig. 10 .

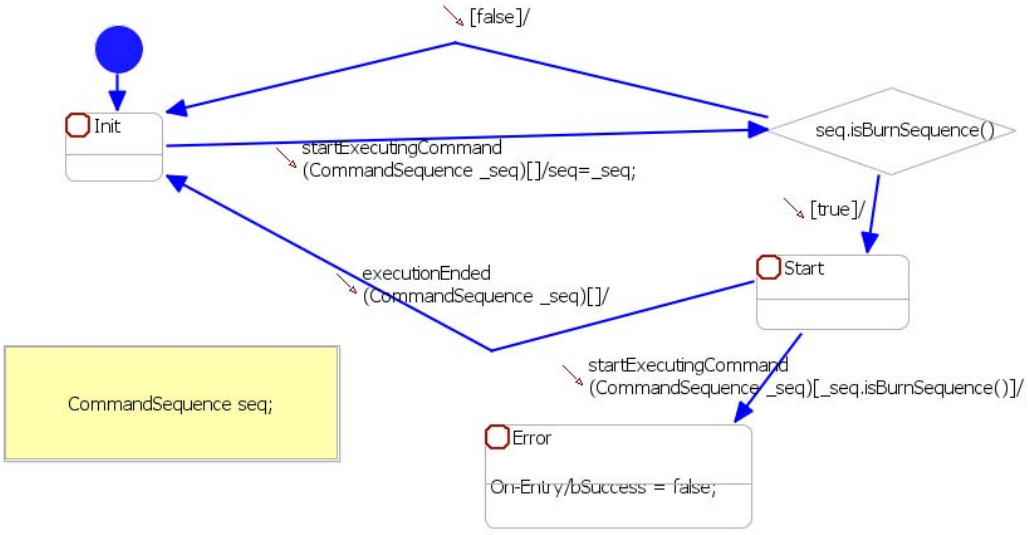


Figure 10. A statechart-assertion for requirement R5.

Consider the *ConstrantRate+ZPitch* AD of Fig. 8 again. As discussed in section C2, the *Determine Attitude* and *Determine Attitude rates* activities are performed concurrently; in the extension scenario either activity can perform a *Reset Kalman Filter* activity and then re-enter the main swim lane and start new *Determine Attitude* and *Determine Attitude rates* activities. This is a case of re-entrance. Several issues are unclear as presented in this AD:

1. Is the Kalman filter shared between the *Determine Attitude* and *Determine Attitude Rates* activities or does each activity have its own copy.
2. Is it possible for *Determine Attitude* to have an error but not *Determine Attitude Rates*, or visa-versa? If so, suppose for *Determine Attitude* has an error but not *Determine Attitude Rates.* In the extension scenario a *Reset Kalman Filter* activity is performed and then both *Determine Attitude* and *Determine Attitude Rates* are

initiated. Does this mean there are two copies of *Determine Attitude Rates* executing? That is probably not the case, so should the currently executing *Determine Attitude Rates* activity be restarted? What if that activity has already finished its task? Alternatively, what if this activity is left untouched but the Kalman filter is reset while its mid-way in its operation? After-all, it might have used old (possibly stale) Kalman filter states in part of its calculation and zero states elsewhere.

5.4. C4: Order and Precedence

This category is about making commitments to the order of activities prescribed in AD's, SD's, and FSM's. Consider the *Insert Orbiter into Lunar Orbit* AD of Fig. 2a. Note the sequence of three activities: *Disable Autonomous Fault Protection*, *Slew to Burn Attitude*, and *Perform Lunar Orbit Insertion Burn*, denoted as DAFP, SBA, and PLOIB, respectively. A concern here is whether this order is required, suggested, or just merely allowed; for example, *is SBA allowed to happen without a following PLOIB?* An additional concern is whether any such rule applies to the entire system or only to this part of the mission, in light of the fact that SBA and PLOIB appear in other AD's, such as in *Adjust Trajectory* AD of Fig. 7. A reasonable question is therefore, do DAFP, SBA, and PLOIB always have to always appear in that order (as prescribed in Fig. 2a)? If not, how about a subsequence, such as SBA followed by PLOIB? The answer to those questions, as provided by Subject Matter Experts (SME's), is the following.

1. PLOIB is a special kind of a *burn* activity, one performed specially for the purpose of inserting the orbiter into lunar orbit. With this and any other burn activity, engines perform a burn to achieve a certain kinetic or attitude maneuver ; a s*lew* (SBA) - an adjustment of attitude, is planned to precede a burn of any kind. However, it is possible in most cases, that fault protection preempt the burn activity. In fact, such a possibility of preemption is indicated in the *Adjust Trajectory* AD of Fig. 7, where *Loss of Attitude Knowledge* or *Loss of Attitude Control* trigger an extension fault-protection scenario whereby the orbiter enters *Safe Mode*. Hence follows requirement R6: *during the execution of an "Insert Orbiter into Lunar Orbit command", fault protection must be off during the slew and burn operations.*

   Fig. 11 depicts a statechart-assertion for the composite of requirements R6 and R7.

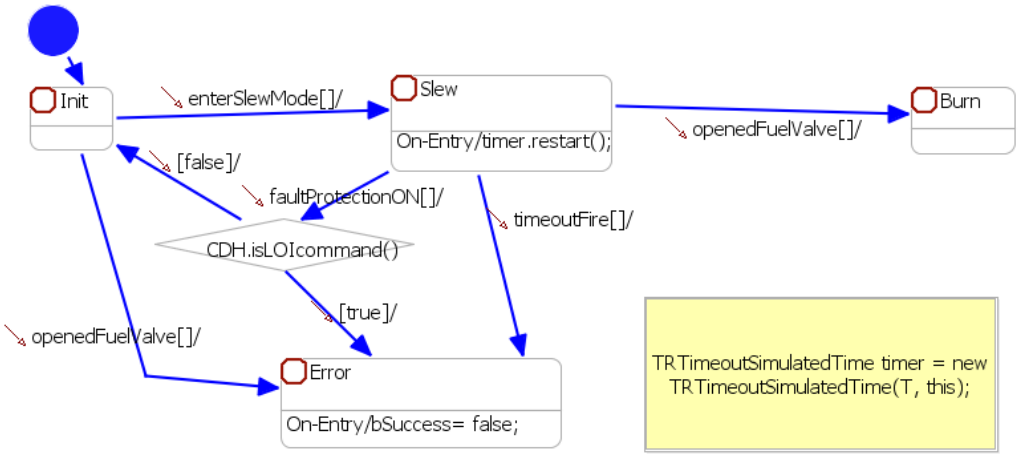


Figure 11. A statechart-assertion the composite of requirements R6 and R7.

Another example from this category is a concern about the *PerformLOIBurn* AD of Fig. 2b. Here, not only that the order listed in the AD is a must, but there are additional constraints as listed in the following NL requirement.

R8: *Once C&DH decides to perform Lunar Orbit Insertion Burn (PerformLOIBurn) the propulsion system must execute the sequence of four activities listed in Fig. 2b precisely in that order, with the following constraints:*

- *a. There must be at least T_PUMP_OPEN_LB seconds warm-up delay between "Turn on Catalyst-bed heaters" and "Open fuel valve".*
- *b. There is an upper bound of T_PUMP_CLOSE seconds between "Close fuel valve" and "Turn off Catalyst-bed heaters".*
- *c. "Perform constant rate pitch maneuver" must be active while the fuel valve is open (with 500 millisecond accuracy tolerance).*

Fig. 12 depicts a statechart assertion for items *a* and *b* of requirement R8; it also includes additional common sense bounds, such as an upper bound on amount of time the pump can be open. Note the event labeled *eventTRdefault*; as discussed in my previous book it is a built-in event whose behavior resembles that of "else", i.e., it fires if none of the other transitions that induce a state change fire. This helps us specify formally that the only acceptable sequence in the AD of Fig. 2b is precisely the one specified in the diagram (the scenario captured by the JUnit test of Listing 5.a), and no other variant be it a sub-sequence or super-sequence, is allowed; hence for example, the assertion prohibits the super-sequence of Listing 5b, in which an *openFuelValve* event happens twice, once before *turnOnCatalystBedHeaters* and once afterwards.

```
public void testMe1() {
      LOIBurnSequence seq = new LOIBurnSequence(1000, 100);
      assertion.startExecutingCommand(seq);
      assertion.incrTime(1);
      assertion.turnOnCatalystBedHeaters();
```

```
        assertion.incrTime(35); // sufficient warm-up time

        assertion.openFuelValve();
        assertion.incrTime(20*60); // burn for 20 min
        assertion.closeFuelValve();
        assertion.incrTime(1); // sufficiently soon therafter
        assertion.turnOffCatalystBedHeaters();
        assertTrue(assertion.isSuccess());
}
```

a. The precise sequence described in the AD of Fig. 2b (with additional timing delays)

```
public void testMe2() {
        LOIBurnSequence seq = new LOIBurnSequence(1000, 100);
        assertion.startExecutingCommand(seq);
        assertion.incrTime(1);
        assertion.openFuelValve(); // a “redundant” event not
                                   // formally prohibited by
                                   // the AD of Fig. 2b.
        assertion.incrTime(1);
        assertion.turnOnCatalystBedHeaters();
        assertion.incrTime(35); // sufficient warm-up time

        assertion.openFuelValve();
        assertion.incrTime(20*60); // burn for 20 min
        assertion.closeFuelValve();
        assertion.incrTime(1); // sufficiently soon therafter
        assertion.turnOffCatalystBedHeaters();
        assertFalse(assertion.isSuccess());
}
```

b. A variant sequence not specifically forbidden by in the AD of Fig. 2b (with additional timing delays)

Listing 5. Validation sequences for the requirement R8.

Fig. 13 depicts a nested statechart assertion for item *c* of requirement R8, using the sub-statechart notation described in [Dr1]. The top-level statechart assertion (Fig. 13a) references the same *checkConcurrentTiming* sub-statechart of Fig. 13b twice.

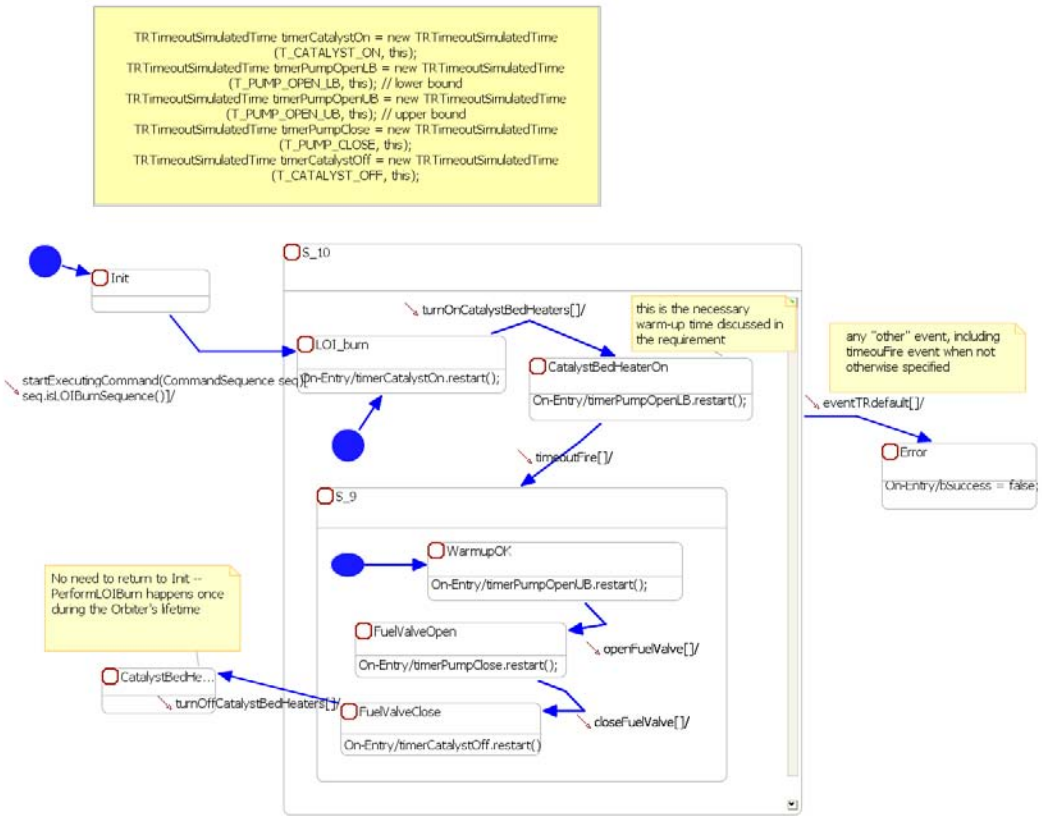
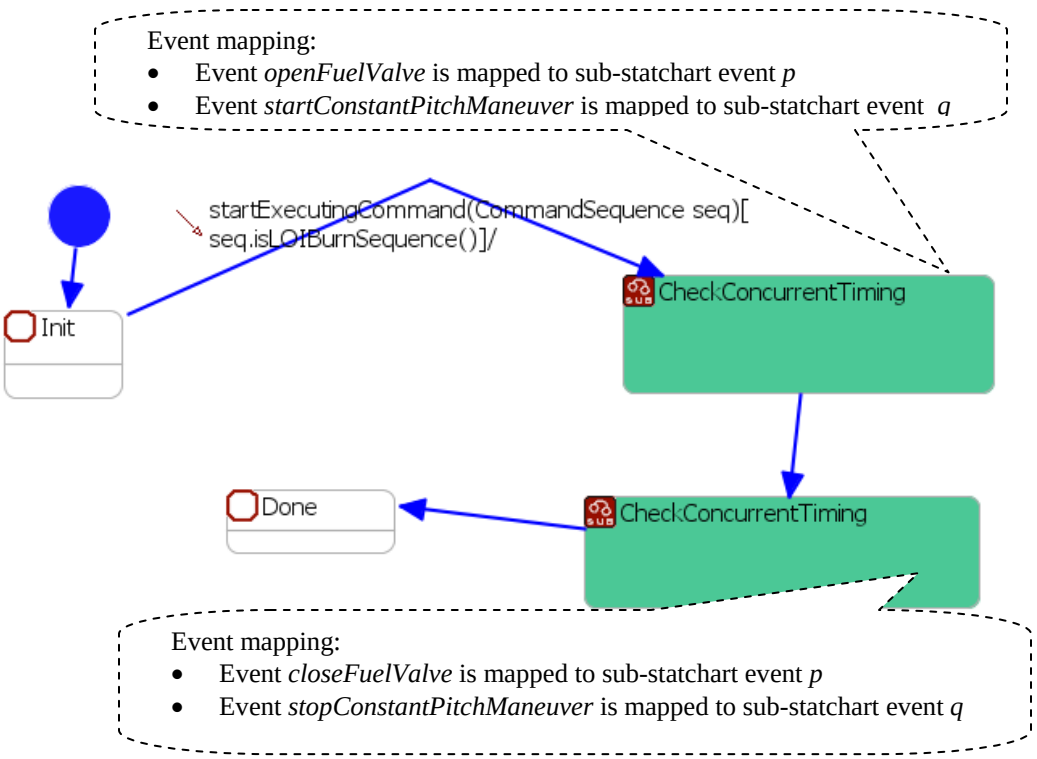
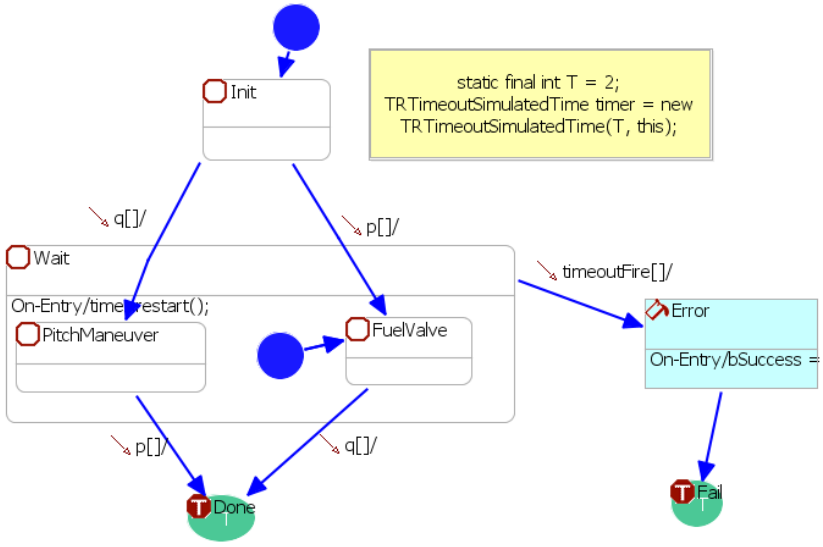


Fig. 12. A statechart assertion for items *a* and *b* of requirement R8.



a. Top level statechart assertion. The sub-statecharts represent a nested assertion of Fig 12.b.



b. The sub-statchart for Fig. 13a.

Fig. 13. A statechart assertion for item *c* of requirement R8.

THIS PAGE INTENTIONALLY LEFT BLANK

6. References

[CTL] E. M. Clarke and E. A. Emerson, “Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic,” Proc. Workshop on Logic of Programs, D. Kozen, ed., LNCS 131, Springer-Verlag, 1981, pp. 52-71.

[Dr1] D. Drusinsky, Modeling and Verification Using UML Statecharts, Elsevier Publishing, 2006.

[DMS] Drusinsky, D. Michael, J. B., and Shing, M. T., A Visual Tradeoff Space for Formal Verification and Validation Techniques, IEEE Systems Journal, Vol. 2, No. 4, Dec 2008, pp. 513-519. ISSN: 1932-8184.

[LTL] D. Harel, A. Pnueli, H. Kugler, Y. Lu, and Y. Bontemps, “Temporal Logic for Scenario-Based Specifications”, in N. Halbwachs and L. D. Zuck, editors, TACAS, vol. 3440 of LNCS, Springer, 2005, 445–460.

[Ha] D. Harel, “Statecharts: A Visual Formalism for Complex Systems”, Science of Computer Programming 8, 1987, 231-274.

[Ec] http://www.eclipse.org

[Ju] http://www.junit.org/

[GRAIL] http://nasascience.nasa.gov/missions/grail

[GPM] http://gpm.gsfc.nasa.gov/

[Sr] http://www.time-rover.com

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research and Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, California

4. Steven Raque
   NASA IV&V Facility
   100 University Dr.
   Fairmont, West Virginia

5. Doron Drusinsky
   Naval Postgraduate School
   Department of Computer Science
   Monterey, California